1st Asian-Pacific Summer School on Formal Methods
Course 11: Hoare Logic with Pointers

Virgile Prevosto

CEA List

August 29, 2009

frama C Software Analyzers

Hoare logic

- Hoare triples $\{P\}s\{Q\}$, meaning "*If we enter statement s in a state verifying P, the state after executing s will verify Q*".
- Function contracts, pre- and post-conditions.
- Weakest pre-condition calculus and program verification.
- The *Why* language.

Position of the problem

Memory update

- in “classical” Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[] = i;
  i=i+1;
  x[] = i;
 }
```

Position of the problem

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
  int i = 0;
  x[0] = i;
  i=i+1;
  x[1] = i;
 }
```

Position of the problem

Memory update

- in “classical” Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
  int i = 0;
  x[0] = i;
  i=i+1;
  x[1] = i;
```

$x_0 == 0 \wedge x_1 == 1$

```
}
```

Position of the problem

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

$x_0 == 0 \land i + 1 == 1$

Example

```
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[0] = i;
  i=i+1;
  x[1] = i;
 }
```

Position of the problem

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
  int i = 0;
  x[0] = i;
  i=i+1;
  x[1] = i;
 }
```

$i == 0 \land i + 1 == 1$

Position of the problem

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```c
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
  x[i] = i;
 }
```

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays, structures?

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
   int i = 0;
   x[i] = i;
   i=i+1;
   x[i] = i;
 }
```

$x_0 == 0 \wedge x_1 == 1$

Position of the problem

Memory update

- in "classical" Hoare logic, variables are manipulated directly
- What happens if we add pointers, arrays,

$$(i + 1 == 0 \Rightarrow$$
$$(i{+}1 == 0 \wedge x_1 == 1))\wedge$$
$$(i + 1 == 1 \Rightarrow$$
$$(x_0 == 0 \wedge i + 1 == 1))$$

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
  x[i] = i;
 }
```

Summary

Functional Arrays

Aliasing

Memory models

Conclusion

Operations

```
type 'a farray
logic select: 'a farray, int -> 'a
logic store: 'a farray, int, 'a -> 'a farray
```

Axioms

```
axiom select_store_eq:
forall a:'a farray. forall i: int. forall v: 'a.
   select(store(a,i,v),i) = v
axiom select_store_neq:
forall a:'a farray. forall i,j: int. forall v: 'a.
   i <> j ->
   select(store(a,i,v),j) = select(a,j)
```

Presentation

Operations

```
type 'a farray
logic select: 'a farray, int -> 'a
logic store: 'a farray, int, 'a -> 'a farray
```

Axioms

```
axiom select_store_eq:
forall a:'a farray. forall i: int. forall v: 'a.
   select(store(a,i,v),i) = v
axiom select_store_neq:
forall a:'a farray. forall i,j: int. forall v: 'a.
   i <> j ->
   select(store(a,i,v),j) = select(a,j)
```

Operations

```
type 'a farray
logic select: 'a farray, int -> 'a
logic store: 'a farray, int, 'a -> 'a farray
```

Axioms

```
axiom select_store_eq:
forall a:'a farray. forall i: int. forall v: 'a.
   select(store(a,i,v),i) = v
axiom select_store_neq:
forall a:'a farray. forall i,j: int. forall v: 'a.
   i <> j ->
   select(store(a,i,v),j) = select(a,j)
```

Correspondence

- array assignment is represented with `store`
- array access is represented with `select`

Example

```
int x[2];

/*@ ensures x[0]==0 &&
              x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
  x[i] = i;
 }
```

Correspondence

- array assignment is represented with `store`
- array access is represented with `select`

Example

```
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
   [i] = i;
```

$$access(x, 0) == 0 \wedge access(x, 1) == 1$$

Correspondence

- array assignment is represented with `store`
- array access is represented with `select`

Example

```
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
  x[i] = i;
```

$$access(store(x, i+1, i+1), 0) == 0 \wedge \ldots$$

Usage

Correspondence

- array assignment is represented with `store`
- array access is represented with `select`

Example

```
int x[2];

/*@ ensures x[0]==0 &&
            x[1] == 1;*/
int main () {
  int i = 0;
  x[i] = i;
  i=i+1;
```

$access(store(store(x, i, i), i+1, i+1), 0) == 0 \wedge \ldots$

Length and Validity of Accesses

Up to now our arrays are infinite: we can access or update any cell.

- Each array has a length
- `select` and `store` have to be guarded
- Use imperative arrays, *i.e.* references to functional arrays

Length

```
logic length: 'a farray -> int
axiom length_pos: forall a: 'a farray. length(a) >= 0
axiom store_length:
forall a:'a farray. forall i: int. forall v: 'a.
length(store(a,i,v)) = length(a)
```

Up to now our arrays are infinite: we can access or update any cell.

- Each array has a length
- `select` and `store` have to be guarded
- Use imperative arrays, *i.e.* references to functional arrays

Length

```
logic length: 'a farray -> int
axiom length_pos: forall a: 'a farray. length(a) >= 0
axiom store_length:
forall a:'a farray. forall i: int. forall v: 'a.
length(store(a,i,v)) = length(a)
```

Length and Validity of Accesses

Up to now our arrays are infinite: we can access or update any cell.

- Each array has a length
- `select` and `store` have to be guarded
- Use imperative arrays, *i.e.* references to functional arrays

Length

```
logic length: 'a farray -> int
axiom length_pos: forall a: 'a farray. length(a) >= 0
axiom store_length:
forall a:'a farray. forall i: int. forall v: 'a.
length(store(a,i,v)) = length(a)
```

Guarded accesses

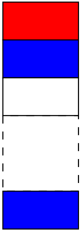
```
parameter select_:
  a:'a farray ref -> i: int ->
  { 0 <= i < length(a) }
  'a reads a
  { result = select(a,i) }
parameter store_:
  a: 'a farray ref -> i: int -> v:'a ->
  { 0 <= i < length(a) }
  unit writes a
  { a = store(a@,i,v) }
```

Dijkstra's dutch flag

Description

Let x be an array whose elements are either `BLUE`, `WHITE`, or `RED`. We want to sort x's elements, so that all `BLUE` are at the beginning, `WHITE` in the middle, and `RED` at the end.

initial state

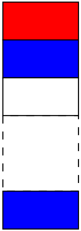


Dijkstra's dutch flag

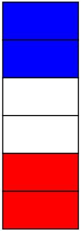
Description

Let x be an array whose elements are either `BLUE`, `WHITE`, or `RED`. We want to sort x's elements, so that all `BLUE` are at the beginning, `WHITE` in the middle, and `RED` at the end.

initial state



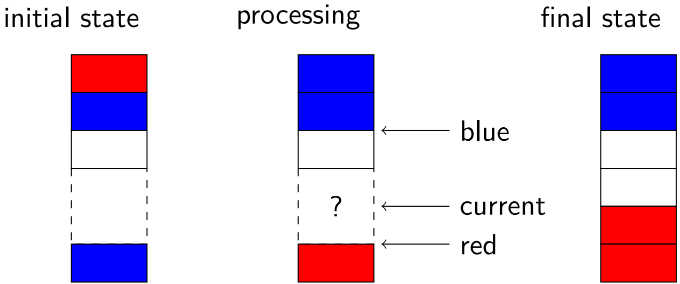
final state



Dijkstra's dutch flag

Description

Let x be an array whose elements are either `BLUE`, `WHITE`, or `RED`. We want to sort x's elements, so that all `BLUE` are at the beginning, `WHITE` in the middle, and `RED` at the end.



Dutch flag in C

```c
typedef enum { BLUE, WHITE, RED } color;

void dutch(color a[], int length) {
  int blue = 0, current = 0, red = length - 1;
  while (current < red) {
    switch (a[current]) {
      case BLUE : a[current]=a[blue]; a[blue]=BLUE;
                  white++; current++; break;
      case WHITE: current++; break;
      case RED  : red--; a[current]=a[red];
                  a[red]=RED; break;
      }
    }
}
```

in Why: colors

```
type color
logic BLUE,WHITE,RED: color

axiom is_color: forall c: color.
  c = BLUE or c = WHITE or c = RED

parameter eq_color: c1:color -> c2:color ->
{} bool { if result then c1 = c2 else c1 <> c2 }
```

```
logic monochrome:
  color farray, int, int, color -> prop

axiom monochrome_def_1:
  forall a: color farray. forall low,high: int.
  forall c: color.
  monochrome(a,low,high,c) ->
    forall i:int. low<=i<high -> select(a,i) = c

axiom monochrome_def_2:
  forall a: color farray. forall low,high: int.
  forall c: color.
    (forall i:int. low<=i<high -> select(a,i) = c) ->
   monochrome(a,low,high,c)
```

```
let flag = fun (t: color farray ref) ->
begin
  let blue = ref 0 in
  let current = ref 0 in
  let red = ref (length !t) in
  while !current < !red do
    let c = select_ t !current in
    if (eq_color c BLUE) then begin
      store_ t !current (select_ t !blue);
      store_ t !blue BLUE;
      blue:=!blue+1;
      current:=!current + 1
    end
...
```

Why Program(cont'd)

```
...

else if (eq_color c WHITE) then
      current:=!current + 1
    else begin
      red:=!red−1;
      store_ t !current (select_ t !red);
      store_ t !red RED
    end
  done
end
```

Function contract

No pre-condition

Post-condition:

```
{ exists blue: int. exists red: int.
    monochrome(t,0,blue,BLUE) and
    monochrome(t,blue,red,WHITE) and
    monochrome(t,red,length(t),RED)
}
```

Don't forget the loop invariant

```
{ invariant
  0<=blue and blue <= current and
  current <= red and red <= length(t) and
  monochrome(t,0,blue,BLUE) and
  monochrome(t,blue,current,WHITE) and
  monochrome(t,red,length(t),RED)
}
```

Proof

Is the program correct?

All proof obligations are discharged by alt-ergo:

```
gwhy dutch.why
```

Further specification

Currently, we have only proved that at the end we have a dutch flag. Other points remain:

- Do we have the same number of blue (resp. white and red) cells than at the start of the function?

Summary

Functional Arrays

Aliasing

Memory models

Conclusion

Assignment Rule

Arrays are not the only objects which reflects poorly in the logic. The assignment rule in Hoare logic:

$$\{P[x \leftarrow e]\}x = e\{P\}$$

contains implicit assumptions:

- Expressions e are shared between the original language and the logic
- We can always find a unique location x which is modified (no alias)

Examples of Problematic Constructions

- Pointers
- Structures
- Casts

- Pointer ∼ base address + index
- Must take care of variables whose address is taken

Example

```
int x;
/*@ ensures
*p == \old(*p) + 1; */
void incr (int* p)
  { (*p)++ }
```

```
parameter x: int farray ref
let incr =
fun (p: int farray ref) ->
{ length(p) >= 1 }
 store_ p 0
   ((select_ p 0)+1)
{ select(p,0) =
    select(p@,0) + 1
  and length(p)=length(p@)
}
```

Pointers

- Pointer $\sim$ base address + index
- Must take care of variables whose address is taken

Example

```c
int x;
/*@ ensures
*p == \old(*p) + 1; */
void incr (int* p)
  { (*p)++ }
```

```
parameter x: int farray ref
let incr =
fun (p: int farray ref) ->
{ length(p) >= 1 }
 store_ p 0
   ((select_ p 0)+1)
{ select(p,0) =
    select(p@,0) + 1
  and length(p)=length(p@)
}
```

Pointers

- Pointer ∼ base address + index
- Must take care of variables whose address is taken

Example

```c
int x;
/*@ ensures
*p == \old(*p) + 1; */
void incr (int* p)
  { (*p)++ }
```

```
parameter x: int farray ref
let incr =
fun (p: int farray ref) ->
{ length(p) >= 1 }
 store_ p 0
   ((select_ p 0)+1)
{ select(p,0) =
    select(p@,0) + 1
  and length(p)=length(p@)
}
```

- Pointer $\sim$ base address $+$ index
- Must take care of variables whose address is taken

Example

```
int x;
/*@ ensures
*p == \old(*p) + 1; */
void incr (int* p)
  { (*p)++ }
```

```
parameter x: int farray ref
let incr =
fun (p: int farray ref) ->
{ length(p) >= 1 }
 store_ p 0
   ((select_ p 0)+1)
{ select(p,0) =
     select(p@,0) + 1
  and length(p)=length(p@)
}
```

```
/*@ ensures x == 1; */
int main ()
 {incr(&x);
  return x}
```

```
let main = fun (_:unit) ->
{ length(x) = 1 and
  select(x,0) = 0 }
begin
  incr x;
  select_ x 0
end
{ length(x) = 1 and
  select(x,0) = 1 }
```

Demo

Aliases

Position of the Problem

In the previous example, we only had one pointer. In practice, programs use usually more than that. What happens if two pointers refer to the same location?

Example

```
/*@ ensures *p == \old(*p + 1) &&
            *q == \old(*q + 1); */
void incr2(int* p, int* q) { (*p)++; (*q)++ }
int x;
/*@ ensures x == 1; */
int main () { incr2(&x,&x); return 0 }
```

Position of the Problem

In the previous example, we only had one pointer. In practice, programs use usually more than that. What happens if two pointers refer to the same location?

Example

```
/*@ ensures *p == \old(*p + 1) &&
            *q == \old(*q + 1); */
void incr2(int* p, int* q) { (*p)++; (*q)++ }
int x;
/*@ ensures x == 1; */
int main () { incr2(&x,&x); return 0 }
```

Aliases

Position of the Problem

In the previous example, we only had one … practice, programs use usually more than that … refer to the same location?

this is true only if p and q are distinct

Example

```
/*@ ensures *p == \old(*p + 1) &&
            *q == \old(*q + 1); */
void incr2(int* p, int* q) { (*p)++; (*q)++ }
int x;
/*@ ensures x == 1; */
int main () { incr2(&x,&x); return 0 }
```

An erroneous why translation

```
parameter x: int farray ref
let incr2 = fun (p: int farray ref) ->
fun (q: int farray ref) ->
begin store_ p 0 ((select_ p 0)+1);
 store_ q 0 ((select q 0)+1) end
{ select(p,0) = select(p@,0) + 1 and
  select(q,0) = select(q@,0) + 1}
let main = fun (_:unit) -> { select(x,0) = 0 }
begin let _ = incr2 x x  in select_ x 0 end
{ select(x,0) = 1 }
```

result

```
Computation of VCs...
File "pointer2.why", line 28, characters 22-23:
Application to x creates an alias
```

An erroneous why translation

```
parameter x: int farray ref
let incr2 = fun (p: int farray ref) ->
fun (q: int farray ref) ->
begin store_ p 0 ((select_ p 0)+1);
 store_ q 0 ((select q 0)+1) end
{ select(p,0) = select(p@,0) + 1 and
  select(q,0) = select(q@,0) + 1}
let main = fun (_:unit) -> { select(x,0) = 0 }
begin let _ = incr2 x x  in select_ x 0 end
{ select(x,0) = 1 }
```

error is here

result

```
Computation of VCs...
File "pointer2.why", line 28, characters 22-23:
Application to x creates an alias
```

Separation Logic

- Extension of Hoare logic dealing allowing to deal with the heap
- introduced by O'Hearn and Reynolds in 2001-2002
- new logic operators:
  - $l \mapsto v$: the heap contains a single location l with value v
  - $e_1 * e_2$: the heap is composed of two distinct parts, verifying e_1 and e_2 respectively

Example

Pre-condition for `incr2`:

$$\exists n, m : int.p \mapsto n * q \mapsto m$$

Separation Logic

- Extension of Hoare logic dealing allowing to deal with the heap
- introduced by O'Hearn and Reynolds in 2001-2002
- new logic operators:
  - $l \mapsto v$: the heap contains a single location l with value v
  - $e_1 * e_2$: the heap is composed of two distinct parts, verifying e_1 and e_2 respectively

Example

Pre-condition for `incr2`:

$$\exists n, m : int.p \mapsto n * q \mapsto m$$

Most Hoare logic inference rules apply to separation logic. A new rule indicates that it is always possible to extend the heap:

$$\frac{\{P\}s\{Q\}}{\{P * R\}s\{Q * R\}}$$

provided the free variables of R are not modified by s.

- Separation logic is a very powerful formalism to deal explicitly with memory.
- Very few tools deal directly with separation logic
- Some of its concepts are incorporated in memory models

Summary

Functional Arrays

Aliasing

Memory models

Conclusion

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.
- ✔ very close to the concrete execution.
- ✔ allows to represent all program constructions.
- ✗ each store can potentially modify something anywhere
- ✗ in practice formulas quickly become untractable.

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.

✔ very close to the concrete execution.

✔ allows to represent all program constructions.

✘ each `store` can potentially modify something anywhere

✘ in practice formulas quickly become untractable.

Memory as array

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.

✔ very close to the concrete execution.

✔ allows to represent all program constructions.

✘ each store can potentially modify something anywhere

✘ in practice formulas quickly become untractable.

Memory as array

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.

✔ very close to the concrete execution.

✔ allows to represent all program constructions.

✘ each store can potentially modify something anywhere

✘ in practice formulas quickly become untractable.

Memory as array

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.

✔ very close to the concrete execution.

✔ allows to represent all program constructions.

✘ each `store` can potentially modify something anywhere

✘ in practice formulas quickly become untractable.

Memory as array

Presentation

In order to deal with pointers, we have to represent somehow the whole memory state of the program in the logic. This is called a memory model.

A first attempt

- See the memory as one big array, with pointers as indices.
- ✔ very close to the concrete execution.
- ✔ allows to represent all program constructions.
- ✘ each `store` can potentially modify something anywhere
- ✘ in practice formulas quickly become untractable.

In order to overcome the scalability issues of the memory-as-array model, more abstract models can be used.

- Split the memory in distinct, smaller arrays, for locations which are known not to overlap.
- For programs with structures, we use an array per field (`x->a` and `y->b` are necessarily distinct).
- Can be extended to distinguish `int` and `float`, `int` and `struct`

✔ gives smaller formulas

✘ some low-level operations (casts, pointer arithmetic) are out of the scope of the model.

Burstall-Bornat Model

In order to overcome the scalability issues of the memory-as-array model, more abstract models can be used.

- Split the memory in distinct, smaller arrays, for locations which are known not to overlap.
- For programs with structures, we use an array per field (`x->a` and `y->b` are necessarily distinct).
- Can be extended to distinguish `int` and `float`, `int` and `struct`

✔ gives smaller formulas

✘ some low-level operations (casts, pointer arithmetic) are out of the scope of the model.

Burstall-Bornat Model

In order to overcome the scalability issues of the memory-as-array model, more abstract models can be used.

- Split the memory in distinct, smaller arrays, for locations which are known not to overlap.
- For programs with structures, we use an array per field (`x->a` and `y->b` are necessarily distinct).
- Can be extended to distinguish `int` and `float`, `int` and `struct`

✔ gives smaller formulas

✘ some low-level operations (casts, pointer arithmetic) are out of the scope of the model.

Memory Regions

- It is possible to go beyond the Burstall-Bornat partition by using some static analysis to identify regions which do not overlap
- Used by the Jessie tool to refine its model
- New preconditions (separation of pointers) that need to be checked

example

```
int a[2];
 void incr2(int* x, int* y) { ... }

int main() {
   incr2(&a[0],&a[1]);
  return 0;
}
```

Memory Regions

- It is possible to go beyond the Burstall-Bornat partition by using some static analysis to identify regions which do not overlap
- Used by the Jessie tool to refine its model
- New preconditions (separation of pointers) that need to be checked

example

```
int a[2];
 void incr2(int* x, int* y) { ... }

int main() {
    incr2(&a[0],&a[1]);
  return 0;
}
```

pre condition: *separated*(x, y)

Memory Regions

- It is possible to go beyond the Burstall-Bornat partition by using some static analysis to identify regions which do not overlap
- Used by the Jessie tool to refine its model
- New preconditions (separation of pointers) that need to be checked

example

```
int a[2];
 void incr2(int* x, int* y) { . .

int main() {
   incr2(&a[0],&a[1]);
  return 0;
}
```

separated(&a[0],&a[1]) holds

Summary

Functional Arrays

Aliasing

Memory models

Conclusion

Summary

- Dealing with memory can be tricky
- Functional arrays play a central role
- Aliases and separation properties
- Need for memory models
- How to do that in practice: see tomorrow